

A METHOD FOR GENERATING LOCALIZABLE MESSAGE CATALOGS FOR JAVA-BASED APPLICATIONS

FIELD OF THE INVENTION

The present invention relates generally to windows-based computer applications, and more particularly to localizable message catalogs for Java-based applications.

BACKGROUND OF THE INVENTION

The increasing use of the Internet and other distributed networks for a variety of purposes, including international commercial, scientific and cultural discourse, makes the ability to readily and reliably produce and support global software increasingly important. Global software, or applications, refers to software that is developed independently of specific languages or geographic regions and capable of being translated into any desired language or region of an end user at run-time.

The internationalization process implies that the software consists of a single set of binaries that operates under all languages, and that the language-sensitive areas in the source code, referred to as the "localizable areas", such as end-user visible strings, data and numeric formats, are stored in external files. The C/C++ programming language, for instance, uses this internationalization model by storing the language-sensitive areas in external files called "message catalogs". Message catalogs are based on the standard

1 defined by the X/Open Portability Guide (XPG) and are thus very attractive for
2 localization. These message catalogs are not source code but are text files,
3 designed for localizability, that allow easy translation of texts into native
4 languages, such as French, Italian, Russian, German or Japanese. The
5 C/C++ programming language "loads" the appropriate language versions of
6 these message catalogs based on the end-user's language at run-time. When
7 a user starts up an internationalized application, the application first checks to
8 see which locale is in use by the user. For instance, when a user runs an
9 application on a German NT server, the user is using the German locale. The
10 locale in use by the user, determined at run-time, will be used by the
11 application for the display text and other user interface elements. If the user
12 changes the locale in use, such as by restarting the desktop environment
13 under another locale, the application will use that other locale to display the
14 texts and other user interface elements.

15
16 This internationalization approach is not available for software written in
17 the portable Java programming language by Sun Microsystems because the
18 Java programming language does not store language-sensitive areas in the
19 source code in message catalogs. The Java programming language provides
20 a framework for developing global applications which allows for translation of
21 end-user visible strings, referred to as "display strings" or "localizable strings,"
22 that may be shown to an intended user and therefore need to be translated for
23 different countries. A Java global application is comprised of a collection of

1 related Java source code files, hereafter referred to as a "package". Each of
2 the Java source files contained in a package may contain display strings. To
3 produce global software, these display strings need to be translated to
4 different countries or languages of the intended users. This translation
5 includes translation of messages, numbers, dates, and currency into a
6 country's conventional formats. Non-display strings, such as comments and
7 Universal Resource Locators (URLs), are used programmatically and are thus
8 not translated.

9
10 As mentioned, the Java programming language does not store display
11 strings in message catalogs but instead stores language-sensitive areas in a
12 source code data structure referred to as a "ListResourceBundle". Basically,
13 a ListResourceBundle data structure provides a way to access display strings
14 according to locale conventions. The ListResourceBundle data structure has a
15 unique identifier to a display string, referred to as a "display string key", that
16 enables "display string value" mapping. A Java ListResourceBundle data
17 structure can be stored in a separate external file and loaded at run-time.

18
19 Figures 1-2 provide an example of how Java programmers use existing
20 Java methodology to internationalize a Java program and then how
21 translators, also called localizers, subsequently localize the internationalized
22 Java program. Referring to the flow chart 10 of **Figure 1**, the first step, shown
23 at Block 12, for internationalizing a Java program is to identify the localizable

1 areas, such as localizable strings, within the Java source code. For purposes
2 of this example only, assume that the Java code of interest is contained in a
3 file named "myApplication.java":

```
4      /* "Display adjustment" and "Volume adjustment" are the */  
5      /* strings that needs to be made localizable. As it appears */  
6      /* in this manner, they cannot be translated. */  
7      myCheckbox1 = new Checkbox("Display adjustment");  
8      myCheckbox2 = new Checkbox("Volume adjustment");
```

9
10 It will be understood that only two strings have been shown in the example for
11 ease of explanation and that one skilled in the art will recognize that Java
12 source code will typically have dozens or even hundreds of strings. The next
13 step, at Block 14, is to create a unique key for each of the identified localizable
14 strings. Assume that the key for the localizable string "Display adjustment" will
15 be "display_adj" and that the key for the localizable string "Volume adjustment"
16 will be "volume_adj". At Block 16, the next step is create a subclass of a
17 ListResourceBundle and override its getContents() method that will contain the
18 localizable string(s). The following is some sample Java code in a file called
19 "myResources.java":

```
20      import java.util.*;  
21      public class myResources extends ListResourceBundle {  
22          public Object[][] getContents() { return contents };  
23          static final Object[][] contents = {  
24              {"display_adj", "Display adjustment"},  
25              {"volume_adj", "Volume adjustment"}  
26          };  
27      };
```

1
2 Then, the locale in the source code must be determined so that the correct
3 language version of ListResourceBundle can be loaded/accessed by the Java
4 program at run-time, as shown in Block 18. Sample Java code that will
5 accomplish this step is shown below and will be added prior to the Java code
6 associated with Block 12 above:

7 /* Obtain the locale information for this application. */
8 /* If this Java program is running in French mode, */
9 /* "getDefault()" will return French. */
10 Locale locale;
11 locale = Locale.getDefault();
12
13 /* Use the locale information (from above) to obtain the */
14 /* locale-specific version of myResources. So, if the locale */
15 /* is set to French, the "getBundle()" will return a French */
16 /* language version of the ListResourceBundle. */
17 public static ResourceBundle rb;
18 rb = ResourceBundle.getBundle("myResources", locale);
19

20 Finally, at Block 20, the localizable string(s) are obtained from the
21 ListResourceBundle. Sample Java code to accomplish this step in the
22 "myApplication.java" file follows:

23 /* Obtain the locale information for this application. */
24 /* If this Java program is running in French mode, */
25 /* "getDefault()" will return French. */
26 Locale locale;
27 locale = Locale.getDefault();
28

```

1      /* Use the locale information (from above) to obtain the */
2      /* locale-specific version of myResources. So, if the locale */
3      /* is set to French, the "getBundle()" will return a French */
4      /* language version of the ListResourceBundle. */
5      public static ResourceBundle rb;
6      rb = ResourceBundle.getBundle("myResources", locale);
7
8      /* Obtain the localizable strings by calling "getString()" */
9      /* and by passing the key for each of the localizable strings. */
10     /* So, if the above calls fetched a French ListResourceBundle, */
11     /* myResources.rb.getString("display_adj") */
12     /* will return the French translation of the string */
13     /* "Display Adjustment" */
14     myCheckbox1 = new Checkbox(myResources.rb.getString("display_adj");
15     myCheckbox2= new Checkbox(myResources.rb.getString("volume_adj");

```

The Java program has thus been internationalized according to the prior art method of Figure 1. Now suppose that the internationalized Java program must be translated or localized to the desired locale. This methodology is illustrated by flow chart 30 of **Figure 2**. The first step, at Block 32, is to obtain the ListResourceBundle for a given Java program that is to be localized. The ListResourceBundle will consist of Java code with "{key, value}" pairs. Sample Java code in the "myResources.java" file that will accomplish this step follows:

```

24     import java.util.*;
25     public class myResources extends ListResourceBundle {
26         public Object[][] getContents() { return contents };
27         static final Object[][] contents = {
28             {"display_adj", "Display adjustment"},

```

```
1      {"volume_adj", "Volume adjustment"}
2      };
3  };
4
```

5 The next step, at Block 34, is to translate only the "value" portion and not the
6 "key" portion of the ListResourceBundle source code data structure, being
7 sure not to modify anything else within the source file. The file may be
8 renamed to indicate the translation language. For instance, the French
9 version of the "myResources.java" file may be titled the "myResources_fr.java"
10 file. Sample Java code in the "myResources_fr.java" file may be as follows:

```
11 import java.util.*;
12 public class myResources extends ListResourceBundle {
13     public Object[][] getContents() { return contents };
14     static final Object[][] contents = {
15         {"display_adj", "Re'glage d'affichage"},
16         {"volume_adj", "Re'glage de volume"},
17     };
18 };
19
```

20 Unfortunately, using a ListResourceBundle data structure presents
21 several disadvantages, as can be seen from the above example. First,
22 ListResourceBundle data structures are source code and are therefore not
23 easily translatable. Translators cannot use any of the "localization tools"
24 applicable for message catalogs and the difficulty becomes distinguishing
25 display strings which need to be translated from non-display strings which do
26 not require translation. The Java code thus has to be combed manually, i.e.

1 line-by-line, to identify the display strings that need to be translated. The
2 display strings must then be changed manually to make it translatable.
3 Needless to say, this process is very time-consuming and inefficient. While
4 mechanisms such as PERL script may be used to assist in this endeavor, the
5 results must still be checked manually.
6

7 Second, writing Java source code that loads and accesses a
8 ListResourceBundle data structure is cumbersome. Additionally, identifying
9 obsolete or new or modified strings that require new translations is difficult
10 using a ListResourceBundle data structure. Modifying the Java source code
11 without translating the accompanying ListResourceBundle data structure will
12 cause the Java program to throw an exception, sometimes referred to as
13 "throwing an exception," similar to a core dump in a C program, for not being
14 able to find a translated string. Finally, modifying an existing non-global
15 application to use ListResourceBundle data structures is difficult because the
16 process of separating the display strings from the Java source code generally
17 cannot be done automatically. It is much easier to build a global program from
18 the ground up rather than attempt to retrofit an existing non-global application
19 because of this difficulty. It is therefore difficult to modify an existing Java-
20 based non-global application to make it capable of using ListResourceBundle
21 data structures.
22

SUMMARY OF THE INVENTION

It is therefore an object of the present invention to provide easily translatable display strings for the Java programming language. It is another object of the present invention to provide a less cumbersome means of loading and accessing a ListResourceBundle data structure. It is yet another object of the present invention to identify obsolete or new or modified display strings that require new translations. Finally, it is an object of the present invention to provide a means of converting an existing non-global application into a global application using ListResourceBundle data structures.

Therefore, according to the present invention, a methodology for generating localizable message catalogs for Java-based applications is disclosed. Message catalogs that are automatically flagged for what needs to be manually translated are generated from a given Java source code file, which can then be used for translation. ListResourceBundle data structures that are compatible with Java's internationalization model are also generated from the message catalogs that were previously generated and manually translated into desired local language(s). The methodology comprises: identifying one or more localizable strings of a Java source code; marking the one or more localizable strings to produce one or more marked localizable strings by inserting a marker into each localizable string of the one or more localizable strings that are function calls to an internationalization tool; extracting the one or more marked localizable strings; storing the one or more

1 marked localizable strings into an external text file, such as a message catalog
2 file; and generating one or more ListResourceBundle data structures from the
3 one or more marked localizable strings stored in the external text file.
4 Extracting and storing the one or more marked localizable strings into one or
5 more text files and generating the one or more ListResourceBundle data
6 structures occurs when the Java source code is compiled. The one or more
7 ListResourceBundle data structures are generated by: determining a current
8 locale language in which the Java source code is running; determining
9 whether the current locale language is a default language; if the current locale
10 language is the default language, returning a marked localizable string of the
11 one or more marked localizable strings; if the current local language is not the
12 default language, determining whether a language-specific version of the
13 marked localizable string that corresponds to the current locale language
14 exists in a ListResourceBundle data structure of the one or more
15 ListResourceBundle data structures that corresponds to the marked
16 localizable string; and if the language-specific version exists, opening the
17 ListResourceBundle that corresponds to the marked localizable string and
18 returning the language-specific version of the marked localizable string from
19 the ListResourceBundle. The one or more marked localizable strings of the
20 Java source code can be translated by: obtaining the external text file
21 containing the one or more marked localizable strings; translating the one or
22 more marked localizable strings stored in the external text file; and generating
23 a ListResourceBundle data structure for the translated one or more marked

1 localizable strings.

2
3 After storing the marked localizable strings into the external text file, the
4 methodology further comprises: generating a new version of the Java source
5 code in a second directory from which an internationalization tool is run;
6 retrieving the marked localizable strings from a ListResourceBundle class; and
7 generating a merged external text file containing the one or more marked
8 localizable strings in the first directory and the second directory.
9 ListResourceBundle files corresponding to the merged external text file with
10 each ListResourceBundle file of the one or more ListResourceBundle files
11 corresponding to a desired language are generated.

12
13 The above methodology is accomplished according to the following:
14 opening an original Java source code file that is stored in a first directory;
15 opening a first message catalog file; copying the first message catalog file to a
16 second message catalog file; creating a modified Java source code file from
17 the original Java source code file in a second directory; reading the contents of
18 the original Java source code file into a buffer; if the buffer contains one or
19 more marked localizable strings, for each marked localizable string of the one
20 or more marked localizable strings comprising: obtaining a message number
21 corresponding to the marked localizable string and replacing the marked
22 localizable string in the buffer with a method call that can obtain the marked
23 localizable string if the marked localizable string is stored in the first message

1 catalog file, or appending the marked localizable string to the second message
2 catalog file and removing a marker from the marked localizable string in the
3 buffer to convert the marked localizable string to a default localized string if the
4 marked localizable string is not stored in the first message catalog file; writing
5 to the modified Java source code file from the buffer; and closing the original
6 Java source code file, the first message catalog file, the second message
7 catalog file, and the modified Java source code file.

8

05443106-100495

1 **BRIEF DESCRIPTION OF THE DRAWINGS**

2 The novel features believed characteristic of the invention are set forth
3 in the claims. The invention itself, however, as well as the preferred mode of
4 use, and further objects and advantages thereof, will best be understood by
5 reference to the following detailed description of an illustrative embodiment
6 when read in conjunction with the accompanying drawing(s), wherein:

7
8 **Figure 1** is a flow chart for internationalizing a Java program, according
9 to the prior art;

10
11 **Figure 2** is a flow chart for translating or localizing the Java program,
12 according to the prior art;

13
14 **Figure 3** is a flow chart that illustrates the methodology for
15 internationalizing a Java program and then translating the internationalized
16 Java program to a desired locale, according to the present invention;

17
18 **Figure 4** is a flow chart that illustrates the MsgHandler.java class,
19 including the "getString" method, according to the present invention;

20
21 **Figure 5** is a flow chart that illustrates the methodology for translating or
22 localizing a Java program, according to the present invention;

Figure 6 is a flow chart that illustrates the methodology of the present invention; and

Figure 7 is a flow chart that illustrates the methodology of the present invention, with emphasis on identifying modified strings that require new translations.

1987	1986	1985	1984	1983	1982
1981	1980	1979	1978	1977	1976
1975	1974	1973	1972	1971	1970
1969	1968	1967	1966	1965	1964
1963	1962	1961	1960	1959	1958
1957	1956	1955	1954	1953	1952
1951	1950	1949	1948	1947	1946
1945	1944	1943	1942	1941	1940
1939	1938	1937	1936	1935	1934
1933	1932	1931	1930	1929	1928
1927	1926	1925	1924	1923	1922
1921	1920	1919	1918	1917	1916
1915	1914	1913	1912	1911	1910
1909	1908	1907	1906	1905	1904
1903	1902	1901	1900	1899	1898
1897	1896	1895	1894	1893	1892
1891	1890	1889	1888	1887	1886
1885	1884	1883	1882	1881	1880
1879	1878	1877	1876	1875	1874
1873	1872	1871	1870	1869	1868
1867	1866	1865	1864	1863	1862
1861	1860	1859	1858	1857	1856
1855	1854	1853	1852	1851	1850
1849	1848	1847	1846	1845	1844
1843	1842	1841	1840	1839	1838
1837	1836	1835	1834	1833	1832
1831	1830	1829	1828	1827	1826
1825	1824	1823	1822	1821	1820
1819	1818	1817	1816	1815	1814
1813	1812	1811	1810	1809	1808
1807	1806	1805	1804	1803	1802
1801	1800	1799	1798	1797	1796
1795	1794	1793	1792	1791	1790
1789	1788	1787	1786	1785	1784
1783	1782	1781	1780	1779	1778
1777	1776	1775	1774	1773	1772
1771	1770	1769	1768	1767	1766
1765	1764	1763	1762	1761	1760
1759	1758	1757	1756	1755	1754
1753	1752	1751	1750	1749	1748
1747	1746	1745	1744	1743	1742
1741	1740	1739	1738	1737	1736
1735	1734	1733	1732	1731	1730
1729	1728	1727	1726	1725	1724
1723	1722	1721	1720	1719	1718
1717	1716	1715	1714	1713	1712
1711	1710	1709	1708	1707	1706
1705	1704	1703	1702	1701	1700
1699	1698	1697	1696	1695	1694
1693	1692	1691	1690	1689	1688
1687	1686	1685	1684	1683	1682
1681	1680	1679	1678	1677	1676
1675	1674	1673	1672	1671	1670
1669	1668	1667	1666	1665	1664
1663	1662	1661	1660	1659	1658
1657	1656	1655	1654	1653	1652
1651	1650	1649	1648	1647	1646
1645	1644	1643	1642	1641	1640
1639	1638	1637	1636	1635	1634
1633	1632	1631	1630	1629	1628
1627	1626	1625	1624	1623	1622
1621	1620	1619	1618	1617	1616
1615	1614	1613	1612	1611	1610
1609	1608	1607	1606	1605	1604
1603	1602	1601	1600	1599	1598
1597	1596	1595	1594	1593	1592
1591	1590	1589	1588	1587	1586
1585	1584	1583	1582	1581	1580
1579	1578	1577	1576	1575	1574
1573	1572				

DESCRIPTION OF THE INVENTION

The method for generating localizable message catalogs for Java-based applications of the present invention provides a mechanism whereby a ListResourceBundle data structure file is automatically generated from an existing Java source code file using an intermediate message catalog file that may be manually translated to a local native language to provide global software that can operate under various native language environments and thus is capable of being used world-wide. This mechanism of the present invention, referred to as the internationalization tool, generates Unix-style message catalogs as intermediate files and then generates ListResourceBundle data structures from those intermediate message catalogs. Without the internationalization tool, the programmer must manually create the ListResourceBundle data structures used by Java to create localizable strings. When a user starts up an internationalized Java-based application, the application determines the locale in use by the user and that locale will be used by the application for the display text and other user interface elements.

Therefore, according to the present invention, a method for generating localizable message catalogs, in which there is one message catalog file per Java package-and not per Java class-is disclosed. First, the present invention generates message catalogs automatically from a given Java source code, which can then be used for translation. Next, the present invention

1 automatically generates the ListResourceBundle data structures from the
2 message catalogs that were derived and translated so that the final result is
3 compatible with Java's internationalization model. Finally, the present
4 invention automatically flags what needs to be translated providing a more
5 efficient means of maintaining a language-specific version of Java-based
6 software after it has been released.

7
8 Referring now to Figures 3 – 7, the methodology of the present invention
9 for internationalizing a Java program and then translating the internationalized
10 Java program to a desired locale is described. Referring first to the flow chart
11 40 of **Figure 3**, the Java programmer identifies the localizable string(s) within
12 the Java source code that make up a Java package at Block 42. These
13 localizable string(s) include previously localized strings, modified strings, and
14 new strings. There is one message catalog per Java package, not per Java
15 class. This step is similar to Block 12 of Figure 1 and sample Java code in file
16 "myApplication.java" follows:

```
17      /* "Display adjustment" and "Volume adjustment" are the      */  
18      /* strings that needs to be made localizable. As it appears */  
19      /* in this manner, they cannot be translated.              */  
20      myCheckbox1 = new Checkbox("Display adjustment");  
21      myCheckbox2 = new Checkbox("Volume adjustment");
```

22
23 The next step, at Block 44, is for the programmer to insert a "><" marker for
24 each of the localizable strings. The file "myApplication.java" is modified in this
25 manner:


```
1      /* "Display adjustment" and "Volume adjustment" are the */
2      /* localizable strings. */
3      myCheckbox1 = new Checkbox("><Display adjustment");
4      myCheckbox2 = new Checkbox("><Volume adjustment");
5
```

6 The localizable display strings "Display adjustment" and "Volume adjustment"
7 of the file "myApplication.java" are easily identified by the "><" inserted by the
8 programmer and are taken out of the source code and stored in one or more
9 external message catalog files for later translation. A representative source
10 version of the message catalog file for the "myApplication.java" file is as
11 follows:

12 \$set 1

13 1 Display adjustment

14 2 Volume adjustment

15
16 It is noted that the message catalog file contains simply the localizable display
17 string and does not contain Java source code, thereby making it easily
18 translatable. This source version of the message catalog file contains
19 messages and is not yet compiled.

20
21 Next, at Block 46, an internationalization tool of the present invention
22 modifies the Java source code to extract the identified localizable string(s),
23 marked with "><", to one or more external, intermediate message catalog files
24 when the Java program is compiled. Strings marked with "><" in the source

1 files are extracted and converted to localizable messages in output file(s), i.e.
2 message catalog file(s). The localized messages in the message catalog
3 file(s) are function calls to an internationalization tool-generated function that is
4 used as an access to ListResourceBundle data structures for Java sources.
5 For Java programs, the internationalization tool converts the message
6 catalogs into ListResourceBundle data structures. This activity occurs
7 automatically when the Java program is compiled. It is transparent to the user
8 and requires no intervention by the Java programmer. As will be described,
9 the internationalization tool also generates the getString function.

10
11 Continuing with the given example, when the above Java program is
12 compiled, the internationalization tool will intervene and modify the Java
13 source code file "myApplication.java" so that it becomes the following:

14 Locale locale;
15 locale = Locale.getDefault();
16 public static ResourceBundle rb;
17 rb = ResourceBundle.getBundle("myResources", locale);
18 myCheckbox1 = new Checkbox(MsgHandler.getString("1","Display
19 Adjustment"));
20 myCheckbox2 = new Checkbox(MsgHandler.getString("2","Volume
21 Adjustment"));
22

23 The internationalization tool automatically generates a new Java class,
24 called "MsgHandler.java", having the definition of one method or function,
25 called the "getString()" method. A representative MsgHandler.java file that is

1 automatically generated by internationalization tool follows:

2 import java.util.*;

3 public class MsgHandler {

4 public static String getString(String key, String defaultMessage) {

5 //SECTION A of CODE

6 if (!initialized) {

7 Locale locale = null;

8 //get the default locale

9 locale = Locale.getDefault();

10
11 //SECTION B of CODE

12 String locale_language;

13 locale_language = locale.getLanguage();

14 if (!(locale_language.equals("en")))

15 { try {

16 rb = ResourceBundle.getBundle("myResources", locale);

17 }

18 catch (Exception ex) {

19 rb = null;

20 }

21 } // if

22 else rb = null;

23 initialized = true;

```

1      }
2  //SECTION C of CODE
3      if (rb == null) return defaultMessage;
4
5  //SECTION D of CODE
6      String str;
7      try {
8          str = rb.getString(key);
9      } catch (Exception ex1) {
10         str = defaultMessage;
11     }
12     return str;
13 }
14 private static ResourceBundle rb;
15 private static boolean initialized = false;
16 }

```

This representative code of the internationalization tool automatically generates the MsgHandler.java class. The MsgHandler.java class consists of one method called "getString". The "getString" method performs various functional steps outlined in Sections A-D of the code and illustrated in the methodology of Figure 4. It is recognized that the particular source code for the "MsgHandler.java" file may be different from that shown above without

1 departing from the spirit and scope of the invention.

2
3 Referring now to Figure 4, methodology 50 of the "MsgHandler.java" file
4 is illustrated. At Block 52, the current locale language being used by the Java
5 program must be determined. Is the program currently running in the English,
6 French, or Japanese languages, for instance? Section A of the
7 MsgHandler.java file above accomplishes this step. Next, at Decision Block
8 54, the inquiry is whether this current language is English, the default
9 language. If it is, then the default English string that was passed into the
10 "getString" method is returned at Block 55; this is accomplished by Section C
11 of the MsgHandler.java file. If the current language is not English, then the
12 flow continues to Block 56. At Block 56, the language-specific version of the
13 ListResourceBundle code is located, as illustrated by Section B of the
14 MsgHandler.java file. Next, the inquiry at Decision Block 58 is whether the
15 language-specific version of the ListResourceBundle code exists. If it does not
16 exist, the flow goes to Block 55 to return the default English string that was
17 passed into the "getString" method; this is demonstrated by Section C of the
18 MsgHandler.java file code. If the language-specific version does exist, then
19 the language-specific localized string from the ListResourceBundle is returned
20 at Block 59, such as demonstrated by Section D of the above code.

21
22 The automatic call performed by the "getString" method has a key
23 benefit for performance. A call to open a ListResourceBundle is relatively

1 expensive in terms of the time required to make a call over a network and the
2 "getString" method provides the advantage of minimizes such calls. Also, from
3 Block 106 of Figure 6, one can see that the internationalization tool will not
4 even call the wrapper function of "getString" if the >< string has not been
5 localized. This too represents a performance improvement because in this
6 case there is no need to check for the locale information and the "getString"
7 method is bypassed.

8
9 In addition to this performance advantage, another advantage that is
10 realized with the "getString" method is the ease with which Java software may
11 be upgraded. This is best explained with an example. Suppose that the
12 internationalized Java program originally contained two localization strings, as
13 in the above example. Those two strings were translated in the French
14 language. When the Java program is executed, the getString method will
15 attempt to retrieve the localized/translated French strings. Now, suppose that
16 the Java programmer adds another "><" string to this program or even
17 modifies an existing "><" string, such as:

18 MyCheckbox1 = new Checkbox("><Display adjustment");

19 MyCheckbox2 = new Checkbox("><Volume adjustment");

20 MyCheckbox3 = new Checkbox("><Color adjustment");

21 It can be seen that a new string for color adjustment as been added.

22
23 When this modified Java program is processed by the

1 internationalization tool, the new "><" string will not be replaced with the
2 getString method call because the internationalization tool knows that this
3 string has not yet been localized as stated in Block 102 of Figure 7. As a
4 result, this modified Java program can execute under the French language
5 even though this new string does not even appear in the ListResourceBundle
6 associated with this French Java program, without throwing an exception.
7 This allows the modified Java source code and ListResourceBundle to be
8 decoupled, a condition that occurs when the ListResourceBundle data
9 structure is not translated at the same time that the Java source code is
10 modified without throwing an exception.

11
12 The above model allows easier upgrade of the Java software. The
13 process has effectively "de-coupled," or separated, two processes: the process
14 of internationalizing the Java program via the "><" indicator, accomplished by
15 Java programmers, and the process of translating the message catalogs,
16 accomplished by the translators. Since these separate processes are de-
17 coupled, they can occur independently. This is an important characteristic
18 when localizing software to any number of different languages for release at
19 the same time-rather than releasing the English version first, followed by other
20 language versions of the code. If these processes were not de-coupled in this
21 manner, the Java program would always have to be in sync with the localized
22 version of the ListResourceBundle data structure. This requirement is
23 particularly onerous in the typical situation in which there is a team of

1 programmers in one country and another team of translators in a different
2 country. It is quite common that the team of translators are almost never on-
3 site with the programmers. Moreover, the costs associated with translation
4 may be prohibitive but, with the present invention, it is no longer necessary to
5 always perform translation of modified or added strings at the time that the
6 Java source code is modified.

7
8 Now that internationalization of the Java program has been performed,
9 the next step is translation of the localizable strings stored in the message
10 catalog file. Referring now to **Figure 5**, flow chart 60 illustrates the
11 methodology of the present invention for translating the localizable string(s)
12 and how the localized ListResourceBundle is generated. At Block 62, the
13 message catalog file associated with the Java package of related Java source
14 code files is obtained. As previously mentioned, it does not contain source
15 code, containing only the display strings, and is therefore readily translatable.
16 Next, at Block 64, the localizable display strings in the message catalog file
17 are translated. Continuing with the above-described message catalog for the
18 "myApplication.java" file, the French translated version of the message catalog
19 is:

20 \$set 1
21 1 Re'glage d'affichage
22 2 Re'glage de volume
23

24 Following the translation, the internationalization tool will again intervene and

1 process the translated message catalog to automatically generate the
2 translated version of the ListResourceBundle Java code, at Block 66. Sample
3 Java code in file "myResources_fr.java" automatically generated by the
4 internationalization tool would be:

```
5 import java.util.*;  
6 public class myResources_fr extends ListResourceBundle {  
7     public Object[][] getContents() { return contents };  
8     static final Object[][] contents = {  
9         {"1", "Re'glage d'affichage"},  
10        {"2", "Re'glage de volume"},  
11    };  
12 };  
13
```

14 As previously discussed, the Java programming language uses
15 ListResourceBundle to create localizable strings. The internationalization tool
16 generates X/Open Portability Guide style message catalogs as intermediate
17 files from which ListResourceBundle may be generated.
18

19 A more detailed example of the methodology of the present invention is
20 presented in the flow diagrams of Figures 6-7. Consider for purposes of this
21 example that the internationalization tool is referred to as "Msgs." Referring
22 now to flow diagram 70 of **Figure 6**, Blocks 72-84 illustrate in greater detail the
23 methodology of the present invention for implementing Block 46 of Figure 3.
24 After identifying localizable strings and inserting "><" for each identified string
25 as shown at Blocks 42 and 44 of Figure 3, the internationalization tool Msgs is

1 run on each of the Java class files that contain the "><" localizable strings to
2 extract the localizable strings and generate a new version of the Java class file
3 in a new directory as shown at Blocks 72 and 74. At Block 72, Msgs extracts
4 the "><" strings into an intermediate message catalog, and at Block 74, Msgs
5 generates a new version of the Java class file in the directory from which Msgs
6 is run. Msgs is executed in the following manner:

```
7      Msgs -J -c $NLS_CAT -d $PRENLS_DIR -n $NLS_DIR -D . -K \  
8          -p $PACKAGE_NAME $SOURCE_DIR/$SOURCE_FILE
```

9 in which -J specifies "Java mode" and -c \$NLS_CAT specifies the name for
10 the message catalog. -d \$PRENLS_DIR specifies the path to the pre-nls
11 directory and this path may be a relative path; the pre-nls directory specified
12 by \$PRENLS_DIR contains the original message catalog file that was translated
13 before. -n \$NLS_DIR specifies the path to nls/C directory and this
14 specification may also be a relative path. nls/C is a directory that will contain
15 a new message catalog file that is effectively the entire contents of pre-
16 nls/\$NLS_CAT file with all of the new and modified messages that were found
17 in the Java program. This new message catalog is called nls/C/\$NLS_CAT
18 file. Once this new nls/C/\$NLS_CAT file is translated, it will replace
19 \$PRENLS_DIR/\$NLS_CAT file. -D path specifies the destination for the output
20 of the modified version of the Java class. The programmer should take care to
21 not overwrite the original Java class file by specifying a different location than
22 the original source location. -K is for the multiple message mode and must be
23 specified. -p \$PACKAGE_NAME specifies the complete name of the Java

1 package to which the particular Java class belongs. \$SOURCE_DIR is the path
2 to the directory that stores the localizable Java class files identified by "><".
3 This directory should be different from the directory from which the Msgs
4 internationalization tool is executed, i.e. the value being passed to the -D
5 option. \$SOURCE-FILE is the name of the Java class file containing the
6 localizable "><" strings. The execution of Msgs by the foregoing commands
7 generates files in the \$NLS_DIR directory and also rewrites a modified version
8 of the .java file.

9
10 At Block 76, from within the same \$NLS_DIR directory, the
11 MsgHandler.java file is generated. MsgHandler.java file will be used by
12 the Java code to retrieve the "><" localizable strings from a
13 ListResourceBundle class. Please refer to Figure 4 and the foregoing
14 discussion for a detailed description of the MsgHandler.java file. The Msgs
15 internationalization tool is executed in the following way to generate the
16 MsgHandler.java file:

```
17 Msgs -J -g -c $NLS_CAT -d $PRENLS_DIR -n $NLS_DIR \  
18 -P $PACKAGE_NAME -B $BUNDLE_NAME
```

19 in which -J specifies "Java mode", -g specifies the "./MsgHandler.java"
20 generate function, -c \$NLS_CAT specifies the name for the message catalog
21 described above, -d \$PRENLS_DIR specifies the path to the pre-nls directory
22 described above, -n \$NLS_DIR specifies the path to the nls/C directory
23 described above, -p \$PACKAGE_NAME specifies the complete name of the

1 Java package to which this Java class belongs as described above, and -b
2 \$BUNDLE_NAME specifies the name to be used for the ListResourceBundle
3 class that Msgs will generate (for instance, "\$NLS_CAT"Res). The foregoing
4 code will cause Msgs to generate a new file called MsgHandler.java as
5 shown at Block 76.

6
7 Next, at Block 78, the Msgs internationalization tool generates a merged
8 intermediate message catalog file. Msgs is executed in the following manner:

9 `Msgs -J -M -c $NLS_CAT -d $PRENLS_DIR -n .`

10 In which -J specifies "Java mode", -M specifies a merge mode, -c \$NLS_CAT
11 specifies the name for the message catalog (described above), -d
12 \$PRENLS_DIR specifies the path to the pre-nls directory described above, and
13 -n specifies the current directory. This language will cause the Msgs tool to
14 generate a message catalog-like file called \$NLS_CAT.1 that contains the
15 merged messages from pre-nls/\$NLS_CAT.1 and the files in the current
16 directory. At Block 80, the intermediate file nls/C/\$NLS_CAT.1 generated by
17 the Msgs tool is now available for localization, i.e. translation into another
18 native language as shown at Block 64 of Figure 5.

19
20 At Block 82, the programmer changes to the \$NLS_DIR directory from
21 which the Msgs tool generates a ListResourceBundle file from the intermediate
22 message catalog file. As an example, MsgsP, a Perl script and not the Msgs
23 executable, is executed in the following manner:

1 MsgsP_ \$NLS_CAT.1 \$BUNDLE_NAME \$PACKAGE_NAME
2 In which \$NLS_CAT specifies the name for the message catalog (described
3 above), \$BUNDLE_NAME specifies the ListResourceBundle class name
4 (described above), and \$PACKAGE_NAME specifies the Java package name to
5 which this ListResourceBundle belongs. MsgsP is run for each translated
6 (localized) version of the message catalog. For instance, if you have a
7 Japanese version of the message catalog stored in the nls/japanese file,
8 MsgsP is run on nls/japanese/\$NLS_CAT.1 as well.

9
10 Finally, at Block 84 all of the .java files are compiled, including the
11 MsgHandler.java and \$BUNDLE_NAME.java files.

12
13 The flow diagram 90 of **Figure 7** further illustrates the methodology of
14 the present invention, with particular emphasis on identifying modified strings
15 that require new translations. At Block 92, the original Java file is opened and
16 next, at Block 94, the pre-nls message catalog file is opened. At Block 95, the
17 pre-nls message catalog file is copied to the nls message catalog file. A
18 modified Java file is created (see Step 74 of Figure 6) at Block 96. At Block 98
19 the contents of the original Java file are read into a buffer or other temporary
20 storage area of a computer on which internationalization tool Msgs is running.
21 Next, at Decision Block 100, the inquiry is whether the string from the original
22 Java file read into the buffer contains "><" to indicate that it has been
23 previously identified as a localizable string. If yes, then the inquiry, at Block

1 102, is whether the "><" string exists in the pre-nls message catalog file. If no,
2 then the string is appended to the nls version of the message catalog file,
3 typically at the end of the message catalog file, at Block 104 and the "><" are
4 removed in the buffer at Block 106 to leave the default English string. If yes,
5 the message number associated with the string is obtained at Block 108 and
6 this "><" string is replaced in the buffer with a method call to "MsgHandler-
7 getString()" that can obtain the localized string at Block 110. The flow
8 continues to Block 112.

9
10 Also from Decision Block 100, if the buffer does not contain a "><"
11 string, the modified Java file is written from the buffer at Block 112. Decision
12 Block 112 inquires as to whether there are any more strings in the buffer. If
13 the end of the file has not been reached, then the flow returns to Block 98 from
14 Decision Block 114. If the end of the file has been reached, however, the flow
15 continues to Block 116. Finally, the original Java file, the pre-nls and nls
16 message catalog files, and the modified Java file are closed at Blocks 116-
17 120.

18
19 An advantage of the methodology of Figure 6 is illustrated by Blocks 104
20 and 105. When a "><" string does not already exist in the message catalog
21 specified by \$NLS_CAT and stored in \$PRENLS_DIR, this indicates that the
22 string is new or has been modified and hence it has never been localized.
23 This new, or modified, string is appended to the bottom of the copied version

1 of the message catalog \$NLS_DIR/\$NLS_CAT. It is this
2 \$NLS_DIR/\$NLS_CAT file, and not the \$PRENLS_DIR/\$NLS_CAT file, that is
3 actually sent to the translators for translation. Because all of the new and/or
4 modified strings are appended to the bottom of the file, the translators can
5 easily identify which strings to translate, thereby significantly speeding up the
6 translation effort.

7
8 The present invention solves a number of problems associated with
9 internationalization of Java-based software and applications. First, the present
10 invention generates message catalogs automatically from a given Java source
11 code, which can then be used for translation. The automatic generation of
12 message catalogs from Java source code avoids having to translate
13 ListResourceBundle data structures which are difficult to translate. Next, the
14 present invention automatically generates the ListResourceBundle data
15 structures from the compiled or processed version of the message catalogs
16 that were generated and translated so that the final result is compatible with
17 Java's internationalization model. Additionally, the present invention
18 automatically modifies the Java source code so that it loads and accesses
19 ListResourceBundle data structures, thereby overcoming the cumbersome
20 prior art procedure of having to write Java code that loads and accesses
21 ListResourceBundle data structures.

22
23 The methodology of the present invention thereby provides a number of

1 advantages over the prior-art methodology. The method is transparent to the
2 user and provides a greatly needed degree of flexibility and consistency in
3 internationalization of Java-based software in the art. The present invention is
4 also time-saving. There is no need to extract strings from the source code for
5 message catalogs; this step is automatically performed by the
6 internationalization tool of the invention. Moreover, the present invention is
7 much faster in that a time-consuming call to open a ListResourceBundle is not
8 executed if a "><" string has not been localized. This provides for faster
9 execution of the Java code. Also, de-coupling the process of translating
10 strings from the process of internationalizing the Java code allows for ease of
11 maintaining the software. The appending of new and/or modified translatable
12 strings to the end of the \$NLS_DIR/\$NLS_CAT file makes it easy for the
13 translator to find the strings to be translated at some future time.

14
15 While the invention has been particularly shown and described with
16 reference to a preferred embodiment, it will be understood by those skilled in
17 the art that various changes in form and detail may be made therein without
18 departing from the spirit and scope of the invention. While only two strings
19 have been discussed in the examples described above for ease of
20 explanation, one skilled in the art will recognize that there are typically
21 hundreds or even more localizable strings. With this great number of strings
22 typically found in Java source code, it can be seen that the job of translating
23 and maintaining the strings is extremely cumbersome using prior art

1 approaches. The present invention greatly improves the efficiency and ease
2 with which global Java source code may be written and maintained.

09:42:06.100:99